ПРЕПОДАВАНЕТО НА ИНФОРМАТИКА: СПЕЦИФИКА И АЛТЕРНАТИВИ

Бойко Банчев

Институт по математика и информатика – БАН
boykobb@gmail.com

Резюме. *В този текст се обсъждат възможни варианти на това, което се преподава по информатика. Става дума главно за средното училище, но смятам, че изложените идеи са приложими за обучението и на по-млади ученици, и на студенти: обсъждането засяга самата същина на програмирането и донякъде на информатиката като фундаментална дисциплина.*

Ключови думи: *информатика, езици за програмиране*

Информатиката като предмет на преподаване. Сравнение с математиката

По метод, общност на подхода и степен на абстрактност информатиката се родее с математиката. Сходството между двете се проявява и в начина на преподаването им. Както математиката в училище се преподава чрез решаване на задачи, така информатиката се преподава най-вече чрез програмиране. Наред с това са налице и определени разлики, на които трябва да обърнем внимание: те са едно от средствата да получим по-ясна представа какво трябва да цели преподаването на информатика и какво да бъде съдържанието му.

Училищната математика, оправдано или не, се придържа към две избрани области – геометрия (свойствата и отношенията на фигурите) и аритметикаалгебра (тези на числата и функциите). Информатиката не е така улегнала. Можем да приемем, че нейна сърцевина е алгоритмиката, но предметната област на алгоритмиката е неясна (алгоритми над какво?), а начините на изразяване на алгоритми – твърде разнообразни и не дотам опознати (не случайно алгоритмичните езици в активна употреба са стотици).

Налице са и други разлики. В математиката доказването на съществуване на даден обект е смислена, ценна задача. Аналогът в информатиката е посочването на алгоритъм за построяване на обект, а алгоритъмът необходимо е конкретен, следователно и задачата – по-трудна. Освен това решаването на задачи в информатиката, за разлика от традиционната математика, винаги става в контекста на ресурсна обусловеност: алгоритмите и програмите работят с определени обеми от данни и се изпълняват в действително или условно време. Ако продължим аналогията с построяването, то трябва не просто да е налице, а да става в рамките на обозрими количества от използвани данни и време. Нещо повече, често е нужно тези количества да бъдат конкретно посочени.

Решението на математическа задача е строго и подробно само дотолкова, доколкото го изисква обстановката, в която то бива представено. Устно или писмено, почти винаги то нито е напълно строго, нито съдържа всички подробности. Езикът, на който изразяваме решението, е само отчасти формален. Слушащият или четящият допълва липсващата строгост чрез собственото си разбиране на засягания материал.

Решението-програма на задача по информатика няма тази свобода. Езикът на изразяване е пределно формален и конкретен – неизбежно, тъй като именно тези му свойства правят възможно програмите да се тълкуват еднозначно, т.е. да се изпълняват. (Няма значение дали става дума за действително изпълняване на компютър или въображаемо: изпълнимостта е свойство, определено чрез езика, на който е записана програмата.) Колкото до степента на подробност на програмите като решения, очевидно тя също е максимална – недоопределените „програми" не са изпълними и в този смисъл изобщо не са програми.

Както става дума по-нататък, програмирането може да приема различни форми и следва да се разбира възможно най-общо. Независимо от формата, важна негова особеност е инструменталната му обусловеност, която в най-общ вид се състои в наличието на действителна или виртуална изпълнителна среда. Именно последната определя какво може да бъде изпълнявано и в какъв вид то се представя за целта. Понякога тя се свежда до даден универсален език за програмиране, друг път – до един или няколко специализирани езика или до интерфейса на някоя приложна система. Във всички случаи най-важната характеристика на изпълнителната среда е задаваният от нея език за взаимодействие. Когато говорим за действително изпълнение, разбира се трябва да са налице и съответните програми: компилатори, интерпретатори и пр.

Що е програмиране?

Понастоящем нямаме единна система от понятия, които да описват програмите или програмирането. Съществуващите стилове на изразяване – императивен, функционален, логически и др. – са толкова

различни, че е трудно да се намери нещо общо между тях. При това всичките те все още се развиват, значи изменят, а е възможно да открием и други.

Разнообразието от гледни точки за и стилове на програмирането е естествено отражение на многоликостта на явлението пресмятане. Под „пресмятане" тук имам предвид всеки процес на преработване на информация: онова, което добре изразява по-абстрактната латино-английска дума computing. Осмислянето на пресмятането и програмирането в тази му общност е според мен предметът на дисциплината информатика. Този общ поглед бе някога определен като „втора грамотност" [1], а днес бива отново потърсен като „изчислително (алгоритмично) мислене" [5].

Многообразието на програмирането се проявява и в чисто практически план – като множество общоцелеви и специализирани езици за програмиране, а и множество програмни системи, взаимодействието с които изисква или допуска една или друга форма на програмиране.

Използва ли се това многообразие при преподаване на информатика в училище? Изобщо не. Преподаването се свежда до начално обучение по езици за програмиране, които при това са и еднообразни, и концептуално ограничени, а и почти без изключение неудобни за целта по самото си устройство.

Всички използвани езици – Бейсик, Паскал, Си/Си++, Лого – са представители на императивния (команден) стил. Всички с изключение на Logo не допускат непосредствено пресмятане в интерактивна среда. У всички липсват удобни средства за изграждане и работа със съставни структури от данни. Като цяло равнището на изразяване у тези езици отговаря повече на представите за ефективна реализация на компилатор отпреди 40 години, отколкото на нуждите на самото програмиране, а още по-малко – на обучението по него. Казаното не подминава и „новите" Джава, Си-шарп и пр., където изобщо се използват.

Обучението по информатика и програмиране би трябвало да дава представа за алгоритмични и даннови структури и принципи и подходи при изграждането им, а вместо това колабира около синтактичните и други чудатости на незнайно защо избраните езици. Моделирането дори на прости понятия и структури се препъва в многобройните наложени от езика присъщи нему особености, нямащи нищо общо нито с решаваните задачи, нито изобщо със същността на програмирането. До обобщаване, абстрактност на модела и обмисляне на алтернативи е практически невъзможно да се стигне.

Пример за това колко езиковоориентирано вместо насочено към същностни за програмирането понятия е обучението по информатика ни дава изтъкването в програмата по този предмет на конструкции като условни оператори, цикли и масиви. Срещаме ги и в нормативни документи, определящи съдържанието на преподаваното, като че става дума за непоклатими стълбове на информатичната наука. Наистина ли това са фундаментални понятия, без които никакво разбиране на програмирането не е възможно? Разбира се, че не – това са само термини от някои езици за програмиране. Има ред други езици, в които те отсъстват, и това отсъствие никак не пречи; начинът на изразяване в подобни езици е друг, често доста по-ефикасен. Такива езици са например функционалните и логическите, но не само те.

Дори понятието променлива като нещо, което приема различни стойности в хода на програмата, е присъщо само на императивния стил на програмиране. (То отсъства и в математиката, която под „променлива" разбира именувана стойност, но не и нещо, свързано с присвояване.)

Условните оператори и тези за цикли са само особени форми съответно на избиране между няколко възможности и на повтаряне. Изборът и повторността обаче имат и поне по няколко други проявления, а и са еднакво свойствени както на действията, така и на структурите от данни. Понятието масив пък е частен случай на редица, т.е. наредена съвкупност от елементи, която може да расте (без значение от кой край, а и не само от краищата), да се съкращава и преподрежда. В редиците може да се търсят стойности, да се извличат отделни елементи или подредици, от тях да се образуват други редици и т.н.

Учебното съдържание по информатика следва да бъде ориентирано към усвояване на общи принципи и похвати, а не на знания за текущо преобладаващите в индустрията езици за програмиране и технологии. Общо и задълбочено разбиране на присъщите на програмирането структури бихме постигнали чрез по-разнообразен избор на езици за програмиране, представящи различни системи от понятия и стилове на изразяване.

Още една възможност за това е разглеждане на програмирането в широк контекст от гледна точка на предметната му област и използваните средства. Не е нужно да се ограничаваме с използване само на универсални езици и традиционни светове като този на числовите алгоритми. Някои специализирани езици и засега пренебрегвани области – текст, геометрични фигури и композиции, дори структури от абстрактни символи, дават чудесни възможности за творческо и изследователско проникване в света на информатиката.

Част от тези алтернативи се разглеждат в следващия раздел.

Алтернативи

Кои са пътищата за обогатяване и разнообразяване на преподаването по информатика? Аз виждам следните няколко възможности:

- избор на подходящ общоцелеви език за програмиране;
- избор на специализиран език или среда;
- разнообразяване на видовете решавани задачи.

Изборът на задачи е в една или друга степен свързан с този на език за програмиране, а последният може да бъде не само един и дори в някои случаи е уместно да са два или повече заедно.

Сред общоцелевите езици особено подходящи за обучение са функционалните. Те съчетават висока изразителност със забележителна непосредственост и лаконичност. Функционалният стил се развива много интензивно през последните години, което намира израз както във все по-голямата употреба на езици от този вид, така и в навлизане на елементи на функционалност в широкоизползваните императивни езици.

Привлекателни черти на съвременните функционални езици са декларативното описване на действия, разнообразните възможности за боравене с функции, възможността динамично да се образуват сложни структури от данни, включително такива с неопределено голям обем и рекурсивни, богатата типова система и диалоговите среди, позволяващи да се изпълняват програмни фрагменти непосредствено.

У такива езици типовете на стойностите се откриват от транслатора самостоятелно, спестявайки на програмиращия необходимостта от изчерпателни описания. Наред с това подобни типови системи са по-изразителни от тези на императивните езици, включително допускайки обобщени (родови) типове. Например функция, която преброява или пренарежда елементите на някоя структура от данни, да речем списък или дърво, има обобщен тип на аргумента си, тъй като в случая типът на частите на структурата е без значение. Така, ако структурата е примерно списък, функцията е приложима върху списъци с елементи числа, низове, списъци, функции или който и да е друг тип.

Сред най-употребяваните днес функционални езици са Хаскел и МЛ. Следват примери с използване на първия. Най-напред няколко определения на функции от стандартната библиотека на езика:

```
map f xs = [f x| x<-xs]
filter p xs = [x| x<-xs, p x]
signum t | t < 0  = -1
         | t == 0 = 0
         | t > 0  = 1
(f . g) x = f (g x)
```

Първата, map, е преобразовател на списъци. Нейни аргументи са функцията f и списъкът xs. map образува списък от стойности f x – прилаганията на f към x – за всеки елемент x на xs. filter също действа върху списък xs, като образува списък от само онези негови елементи x, за които p x е вярно (p е предикат, т.е. функция с булев резултат). За отбелязване е, че определенията и на map, и на filter са съвсем близки до обичайния за математиката начин на записване, което ги прави почти очевидни дори за незапознати с езика. Същото важи и за функцията signum, която намира „знака на число": -1, 0 или 1 според аритметичната стойност на числото.

Последната определена функция е композиция на функции, f след g. Понеже е зададена като операция (.), и при прилагане композицията се записва между двата си аргумента, а не, както за другите функции тук, пред тях.

Голяма част от силата на функционалния език се дължи на разнообразните начини за образуване на функции и в частност на комбинирането им. Функциите са стойности, често безименни, които, участвайки в изрази, водят до получаване на нови функции. Например map има потенциално два аргумента, но като я приложим частично само към signum – map signum – получаваме в резултат нова функция. Приложена към числов списък, примерно [2,17,-3,8,0,-4], тази функция го преобразува така, както ако направо напишем map signum [2,17,-3,8,0,-4] (получаваме [1,1,-1,1,0,-1]), но „междинната" функция е ценна сама по себе си: тя може да бъде композирана с друга, предадена като аргумент и пр.

Аналогично (<) е операцията „по-малко", тълкувана като функция, а (0<) и (<0) са нейни частични прилагания – функциите, проверяващи дали дадено число е положително и отрицателно. Тогава на свой ред filter (0<) е частично прилагане на filter, даващо функция, която от кой да е числов списък избира само положителните числа. Композицията length.filter(0<) пък е функция, намираща броя на положителните числа в даден списък (length дава дължината на списък).

Можем да обобщим последния получен израз, като на мястото на (0<) поставим произволен предикат p: length . filter p е функция, отговаряща на въпроса „за колко от членовете на даден списък е вярно p?". Да забележим, че тази функция вече е приложима и към списъци не само от числа; елементите им могат да

са всякакви – низове, списъци, *n*-ки, функции …. На получената нова функция можем да дадем име, определяйки напр. `numof p = length . filter p`. Тогава `numof (0<)` дава каквото и `length . filter (0<)` от по-горе, но можем да направим и други полезни прилагания на `numof`: `numof even` е функция, преброяваща четните числа в списък, а `numof (`elem` "xyz")` намира колко пъти в даден низ се среща някоя от последните три букви от латиницата.

Виждаме, че с частично прилагане, комбиниране и обобщаване само на няколко изходни функции, без нито буква в повече от необходимото, получаваме ред полезни функции, от които можем да стигнем и до още много други.

Сравнете това с някой от обичайните за училищната информатика езици за програмиране. Подобни резултати изискват написването на многократно по-голяма програма, повечето от текста в която няма пряка връзка с целта ни, но се изисква от „бюрокрацията" на езика. Всъщност на Бейсик, Паскал, Си или Лого дори не можем да получим резултат толкова общ колкото показания, а на Си++ – почти да, но това изисква прилагане на много тежък езиков апарат и като запис се оказва толкова тромаво, че е практически негодно за четене.

Във всички тези езици няма и следа от показаната по-горе непосредственост и простота! Колко успешно тогава може да бъде преподаването на информатика чрез тях? Ако при образуването дори на прости абстракции се натъкваме на градивни трудности, колко далеч можем да отидем в моделирането на частици от света чрез програмиране?

Конструкцията „определител на списък" ([…]), с помощта на която зададохме функциите `map` и `filter`, има много други приложения. С нейна помощ удобно се изразяват различни зависимости при образуване на списъци, което пък е ценен инструмент за моделиране при решаване на числови и комбинаторни задачи. Следната функция намира всички начини, по които аргументът ѝ *n*>0 се представя като сбор от три различни числа:

```
sums3n = [(a,b,c) | a<-[1..n-2], b<-[a+1..n-1],
                    c<-[b+1..n], a+b+c==n]
```

Например `sums3 12` дава `[(1,2,9),(1,3,8),(1,4,7),(1,5,6),(2,3,7),(2,4,6),(3,4,5)]`, а `length (sums3 100)` – 784.

Като използваме, че две или повече изброявания (<-) в определител на списък водят до образуване на декартово произведение, както при тройките в горния пример, можем да напишем рекурсивна функция за намиране на такова произведение от множество редици:

```
cprod [] = [[]]
cprod (xs:xss) = [x:ps | x<-xs, ps<-cprod xss]
```

Да отбележим, че става дума за какъв да е брой редици, всяка с какъв да е брой членове. Например изразът `cprod ["abc","d","ef"]` е равен на `["ade","adf","bde","bdf","cde","cdf"]`.

Последните две приведени определения дават толкова ясни и непосредствени решения на съответните задачи, че няма нужда да бъдат обяснявани. И в тези случаи обаче решенията на Си и пр. биха били дълги, тромави, а и недобре отговарящи на постановката на задачите.

Специално за учебни цели са разработени диалектите на Хаскел Vital, Pivotal и Helium, които имат усилена интерактивност и дружелюбност, вкл. могат да показват графично съдържанието на структури от данни.

Налице са и други привлекателни езикови алтернативи. Една от тях е системата ГеомЛаб [4], разработена именно за училищно обучение по информатика. ГеомЛаб е функционален език за работа с числа и равнинни фигури. Той има много малък брой вградени конструкции и почти тривиален синтаксис, но при все това дава възможност пълноценно, лаконично и с непосредствено онагледяване чрез фигурни композиции да се изразят редица важни за програмирането понятия. Диапазонът на сложност на решаваните чрез него задачи е много широк и в двете посоки, така че опитен преподавател да може да подбере различни по характер и степен на трудност предизвикателства. Наред с другите, достойнство на тази система е това, че визуалното в нея заема точно подходящото място, без да отвлича вниманието от съществото чрез излишна ефектност и предразполагане към рисуване вместо програмиране.

Често прилаган опростен, но все пак меродавен показател за изразителността на езиците за програмиране е размерът на програмата, която решава дадена задача. Като правило, по-кратка програма означава по-адекватен език. На уебсайта [3], където представям информация за множество езици, дадена задача бива решена на по-голямата част от тях – понастоящем петдесет и четири. Може да се забележи, че т.нар. сценарни (scripting) езици (много от които впрочем също носят белези на функционалния стил) дават къси и ясни решения. Тези езици, особено напр. Руби и Пайтън, предлагат интерактивност, високо семантично равнище и разнообразни приложни библиотеки, поради което би могло с успех да бъдат използвани за обучение по информатика.

Интересен и интензивно развиващ се клас програмни среди са тези за динамична геометрия [2]. Тези от тях, като ГеоГебра и П. и Л. (C.a.R.), които предлагат конструктивен език, дават и прекрасна възможност за обучение по програмиране чрез специализираната област на геометричните построения. Такива програми вече широко се използват в занятията по геометрия, но като среди за програмиране са недооценени.

Редица от широкоизползваните програми, наричани обикновено „инструментални" в операционните системи, като bc, dc, sed и awk са в една или друга степен интерпретатори на специализирани езици за програмиране и също успешно могат да се използват като такива в обучението по информатика. За много задачи те са доста по-подходящи от сега използваните езици. Дори добре развит текстов редактор може да предложи много съдържателни задачи за разпознаване и преобразуване на текст с използване на регулярни изрази.

Въпреки че тук отделих внимание най-вече на инструментите – езици и среди – на програмирането, като ценен за обучението по информатика ресурс не бива да подминаваме разнообразието на задачите, които могат да се поставят и решават. Специализираните езици и среди подсказват специфични за всяка от тях задачи за *построяване* на разнообразни по вид обекти: числови множества, геометрични фигури, текстови и абстрактносимволни структури, шаблони за разпознаване. Намирането на оптимални обекти е важна разновидност на този тип задачи. Построяването на определен обект може да бъде чрез програма или не, а може да се състои и в построяване на програма или вход за програма чрез програма! По-сложен вариант на построителна задача – метапостроение – е разработването на „миниезик" за описване или построяване на даден клас от обекти.

Друг тип задачи са тези за *проверка или откриване на свойства*. Те също могат да бъдат специфични за дадена предметна среда. Важна разновидност е изследването на свойствата на самите програми.

Литература

[1] A.P. Ershov. *Programming, the second literacy*, a talk at the 3rd IFIP and UNESCO World Conf. on Computers in Education, 1981.

[2] B. Bantchev. *A brief tour to dynamic geometry software*. http://www.math.bas.bg/bantchev/misc/dgs.pdf.

[3] B. Bantchev. *Centre for programming language awareness*. http://www.math.bas.bg/bantchev/place.

[4] *GeomLab—exploring computer science*. http://web2.comlab.ox.ac.uk/geomlab.

[5] J.M. Wing. *Computational thinking*. Comm. ACM 49 (2006), No.3, pp.33–35.